AI Agents Need Operating Systems, Not Just Better Models

Why the Next Frontier of AI Reliability Is a Software Engineering Problem, Not a Modeling One

By Nishil Pathak

Production failures in AI agent deployments are, in most cases, software engineering failures caused by the absence of principled runtime infrastructure — not model capability failures. This article argues that today's agent frameworks solve composition problems but leave the runtime layer untouched: process isolation, resource governance, structured error handling, durable state, and provenance capture. Drawing on operating systems design, the saga pattern, runtime verification, and transaction theory, it lays out six reliability properties a mature agent execution environment needs, walks through where frameworks such as LangChain and orchestration engines such as Temporal and Airflow fall short, and closes with concrete engineering steps teams can take today without waiting for a dedicated agent runtime to exist.

Key Takeaways

AI agent reliability failures in production are, in most cases, software engineering failures caused by the absence of principled runtime infrastructure — not model capability failures. Fixing them takes systems engineering, not better prompting or bigger models.

Frameworks and runtimes solve different problems. A framework provides conventions for composing agent behavior; a runtime provides lifecycle management, resource enforcement, fault isolation, and recovery guarantees. Today's agent frameworks are framework-level solutions that leave the runtime layer untouched.

Operating systems design offers abstractions that map directly onto agent execution: process isolation, resource governance, structured error hierarchies, consistent state persistence, and observable execution. Current agent infrastructure provides none of these by default.

Long-horizon agent tasks need saga-pattern compensation, so a partial failure can be recovered from a known checkpoint instead of forcing a full restart — something general-purpose workflow engines can't do cleanly for dynamic agent execution.

Structured provenance — a queryable causal record connecting agent inputs, reasoning steps, and outputs — is non-negotiable for agents operating in regulated industries or making consequential decisions, and it has to be built at the runtime level, not bolted on as application logging.

There's a persistent assumption in AI agent discourse that reliability comes exclusively from better foundation models: scale the parameters, improve the training data, refine alignment. That assumption isn't just incomplete — it's architecturally naive, and it produces production failures that have nothing to do with model quality and everything to do with missing software infrastructure.

Teams building production agent systems are relearning, often the hard way, lessons the operating systems and distributed systems communities worked out decades ago. Software that makes decisions, takes actions, manages resources, and coordinates with other systems in

dynamic environments needs more than a capable component at its center — it needs resource management, fault isolation, recovery mechanisms, provenance tracking, and observable state. In short, something that functions like an operating system for agent execution.

The gap in agent engineering right now isn't a modeling gap — it's a systems gap, and closing it means drawing deliberately on operating systems design, distributed systems theory, and runtime verification, rather than waiting for the next generation of models to absorb these properties into their weights.

The Agent Deployment Reality

Set benchmark leaderboards aside and look at what agents actually do in production. A serious engineering deployment isn't a chatbot with tools bolted on. It's a program that perceives an environment, holds state or memory, chooses from a repertoire of actions, executes those actions with real consequences in external systems, and does this repeatedly over long stretches of time with minimal supervision. That's close to what operating systems researchers would call a long-running process in a resource-constrained, multi-tenant environment.

None of the resulting engineering problems are exotic — process isolation, resource accounting, scheduling fairness, error propagation, state recovery. What's new is that the decision-maker at the center is a probabilistic model instead of deterministic code, and that single change reshapes the reliability calculus in ways existing infrastructure was never built to handle.

The scenarios below are composites: patterns I've observed recur across multiple production agent deployments, reconstructed to illustrate the failure mechanism rather than to report on a single named incident. Take a representative case: an agent tasked with auditing a company's software dependencies, flagging vulnerable packages, and filing remediation tickets. It runs for several minutes, querying package databases, parsing manifests, reasoning about transitive dependencies, and writing to an external system. Partway through, a network partition returns a malformed response from one dependency query. With no structured contract around its tool calls, the agent treats the bad data as valid and keeps going. The tickets it files are wrong — they cite vulnerabilities that don't exist and miss ones that do.

What actually broke here wasn't the model; it reasoned exactly as trained. What was missing was a runtime layer able to catch anomalous tool output, enforce data contracts, manage partial failure, and stop the agent's decisions before they hit production systems. That's a software engineering failure with a software engineering fix.

Not every deployment carries this much exposure immediately — short-horizon agents with narrow tool access and human review before any write are considerably safer. The point isn't that every agent needs a full runtime layer on day one. It's that as tasks get longer, tool access broadens, and human oversight thins out, skipping runtime infrastructure gets more expensive and failures get harder to unwind.

- **Practical Engineering Takeaways:** Watch for these signs that your infrastructure has been outgrown — agents writing to external systems with no pre-execution validation, incidents that require correlating logs across services just to reconstruct what happened, or failed runs with no recovery path besides starting over. The most common mistake is mistaking prototype reliability for production readiness: agents that behave on clean test data routinely fail once tool responses get inconsistent and environmental state stops being predictable.

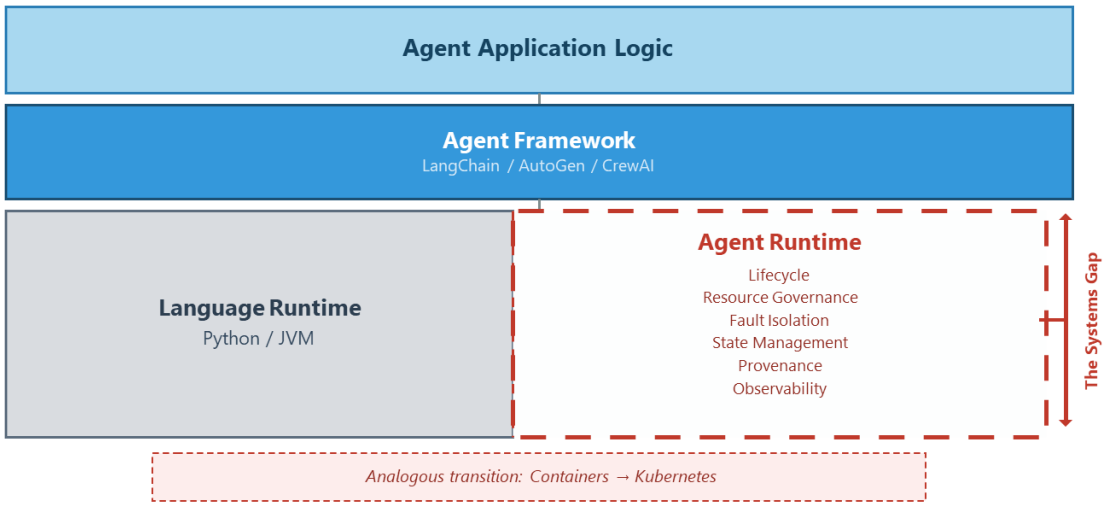

Figure 1. Current AI agent stacks provide a framework layer for composition but leave the runtime layer entirely absent. The reliability gap exists between what frameworks provide and what a production execution environment requires, mirroring the gap that existed in container deployments before Kubernetes.

Why Frameworks Aren't Enough

The instinctive answer from many teams is to point at the growing list of agent frameworks — LangChain, LlamaIndex, AutoGen, CrewAI, Semantic Kernel — which handle composing behavior, managing tool calls, and orchestrating multi-agent workflows. These are genuinely useful, but they're frameworks, not runtimes, and in production that distinction matters a great deal.

A framework is a set of conventions: it shapes how you structure code and standardizes interfaces. A runtime is an execution environment: it manages the lifecycle of running programs, enforces resource limits, handles exceptions at the system level, isolates concurrent workloads, and guarantees behavior around process execution. It's the difference between a web framework and an operating system. Django tells you how to structure a web app; Linux decides whether it runs, for how long, with what resources, and what happens when it crashes.

When a tool call blocks indefinitely, there's no infrastructure-level timeout. When an agent spirals into unbounded memory use, no resource governor steps in. When a three-week-old incident needs an action trace for audit, there's often no provenance record to consult. These frameworks solve composition problems reasonably well; the issue is assuming that's sufficient for production reliability.

The Kubernetes analogy fits well. Before Kubernetes, teams ran containers with shell scripts and homegrown orchestration — the containers worked fine, but the gap was in lifecycle management: health checks, resource enforcement, scheduling, recovery. Kubernetes didn't make containers smarter; it supplied the runtime layer that made them operable at scale. Agents need the same evolution.

- **Practical Engineering Takeaways:** Before building a custom runtime, see how far four targeted controls get you on top of your existing framework: structured output validation on every tool call, a hard step-count ceiling enforced by the orchestrator, a circuit breaker on external API calls, and a dry run before any write. None of these replace a runtime layer —

they're a starting position while you build toward one. A runtime adds real operational complexity, and for low-consequence internal tooling with heavy human oversight that overhead may not be worth it yet. For agents with external write access or compliance obligations, the risk of skipping it outweighs the cost of building it.

The Missing Abstractions

Every abstraction operating systems give conventional processes has a real analog in agent execution. Laying them out side by side shows how wide the current gap is.

- **Process isolation and sandboxing.** Operating systems keep a misbehaving process from corrupting another's resources; for agents, the equivalent is action isolation. Picture a support agent and a billing agent sharing a memory backend — a bug in the support agent's write logic could corrupt billing context for unrelated sessions. Container runtimes solve this with namespace isolation; nothing equivalent exists by default for agents today.
- **Resource management.** Long-running agents consume serious amounts of tokens, compute, and API quota. On a shared platform, one agent chewing through a large document collection can exhaust an organization's rate limit and take down every other agent in the fleet. Kubernetes handles this with resource requests and limits; agents need equivalent governance.
- **State management and persistence.** When an agent analyzing a database migration crashes forty minutes into a ninety-minute task, it usually restarts from zero, re-paying token costs and re-running tool calls that already succeeded. Agent state needs the consistency guarantee operating systems give through filesystem abstractions.
- **Error handling and recovery.** Current agent systems mostly don't distinguish between a transient HTTP 429 worth retrying with backoff, a persistent 403 that needs escalation, and a semantic inconsistency in the agent's own reasoning that calls for re-planning. Blur those together, and you get agents that retry unretryable errors until they burn through their budget, or silently swallow errors that should have stopped execution.
- **Observability and auditability.** When a production agent misjudges a security alert during an autonomous incident-response run and escalates something minor as critical, figuring out why means forensically correlating log files and guessing at the agent's context window at the time. That's not debugging — it's archaeology.
- **Practical Engineering Takeaways:** If you can only tackle one of these now, tackle error classification. Give every tool at least three error classes — retryable, non-retryable, requires-human-review — and encode that in the tool wrapper, not the prompt. The most common and costliest mistake in this space is treating retry logic as fault tolerance. Retry covers one failure mode; it does nothing for partial success, semantic errors in tool output, or state drift after a crash.

Figure 2. Each foundational operating system abstraction has a direct analog in the agent execution domain. The right column represents capabilities that must be engineered deliberately into agent infrastructure. None are provided automatically by current frameworks.

Why the Reliability Challenge Is Structurally Different

A conventional process is deterministic given its inputs and code. An agent's decisions come from probabilistic inference across billions of parameters, so identical inputs can produce different outputs from run to run. That non-determinism is the source of an agent's flexibility, and the reason static verification techniques don't carry over cleanly.

This is where runtime verification becomes directly relevant [1]. Instead of checking code statically, runtime verification watches a system's actual behavior against defined properties and steps in when it drifts. Applied to agents, that means specifying invariants over actions, pre- and post-conditions for tool calls, and constraints on outputs, then checking them as the agent runs. A violation triggers an intervention: pause execution, request human review, roll back completed actions, or re-plan from a known-good checkpoint.

Long-horizon tasks raise a second structural issue that the saga pattern addresses well [2]. A saga breaks a long-running operation into local steps, each paired with a compensating action, so that a failure at step five of twelve can be unwound in reverse rather than left as an inconsistent mess requiring manual cleanup. An agent running a multi-stage code review, test generation, and pull request workflow can define compensating actions at each stage.

Temporal's durable execution and activity retry policies address part of this [3]. The limitation isn't the engine — it's that Temporal assumes a statically defined workflow graph. You can encode dynamic agent behavior by treating the planning loop as an activity that emits sub-activities on the fly, but that takes real engineering investment and produces workflows that are harder to inspect than conventionally structured ones.

- **Practical Engineering Takeaways:** When you have microservices running on Temporal, agent workflows are a natural next step. The key question is how often to save state: save after every LLM call and you lose less progress when things go wrong, but at the cost of much more

memory. A good middle ground is saving state after each tool call completes — you avoid the tedium of saving after every request, but still have somewhere to roll back to. Verification adds its own overhead, and someone has to specify what to check and keep those checks current; start simple, disallowing only the obviously unsafe actions, and add more verifications once the agent is doing useful work.

Where Do Workflow Tools Fall Short?

Running agents inside orchestration systems like Apache Airflow, Temporal, or Prefect beats relying on application-level code alone, but the limitation is architectural: these engines were built for workflows with a control-flow graph fixed before execution starts, while an agent's execution graph forms at runtime as it discovers what to do next. Airflow needs the full task graph up front, so when an agent fails inside an Airflow task, Airflow tells you the task failed — not which reasoning step broke, which tool call returned bad output, or what the agent's context looked like at that moment.

Temporal handles dynamic execution better, but it was designed for human-authored workflow code, not for a workflow whose next step depends on a probabilistic model's output. Replaying a Temporal workflow for debugging assumes the same activity sequence reproduces the same result — that assumption doesn't hold when step six depends on the LLM's output at step five, without extra work like output caching or deterministic sampling.

- **Practical Engineering Takeaways:** Use Temporal for outer orchestration, but keep the agent's reasoning loop as a single activity with its own internal checkpointing, writing state to external storage at each tool call boundary. Every added layer of orchestration is more operational overhead and another place things can break. A five-step agent with human review at the end probably doesn't need Temporal; a multi-stage, unattended DevOps agent with production write access almost certainly does.

The Provenance Problem

Provenance means a complete, structured, auditable record of why the agent made the decisions it made — not just what it decided. That's different from logging: logs record events in order, while provenance captures the causal links between inputs, reasoning steps, and outputs, so the whole decision process can be reconstructed later [4, 5].

Consider a security remediation agent scanning for misconfigured IAM policies. Over a two-hour window, it applies 47 changes; a later audit finds that three of them introduced new vulnerabilities instead of fixing existing ones. Without structured provenance, tracing which inputs drove those specific decisions means manually reconstructing the agent's likely context at the time — slow and unreliable. With a provenance graph, you query it directly, pull the exact tool outputs behind the flawed decisions, and get to root cause in hours instead of days.

In regulated industries, the stakes are higher still. Financial firms deploying agents for credit decisions or regulatory reporting must be able to explain automated decisions to auditors; healthcare organizations face similar obligations. There, provenance isn't a nice-to-have — it's a compliance requirement, and skipping it creates legal exposure many organizations haven't fully priced in.

A flat log of tool calls isn't a provenance record. A real one causally links each action to the reasoning step that produced it, the tool outputs that informed that reasoning, and the input context at that decision point — that causal structure is what makes it queryable instead of just archival.

- **Practical Engineering Takeaways:** Short of building a full provenance graph, two changes go a long way: give every agent run a unique execution ID propagated through every tool call, LLM invocation, and external write, and log the agent's input context at each major decision point, not just the final output. If your incident postmortems ever include the phrase "we think the agent probably saw," that's a provenance gap that will cost more to fix after an incident than before one.

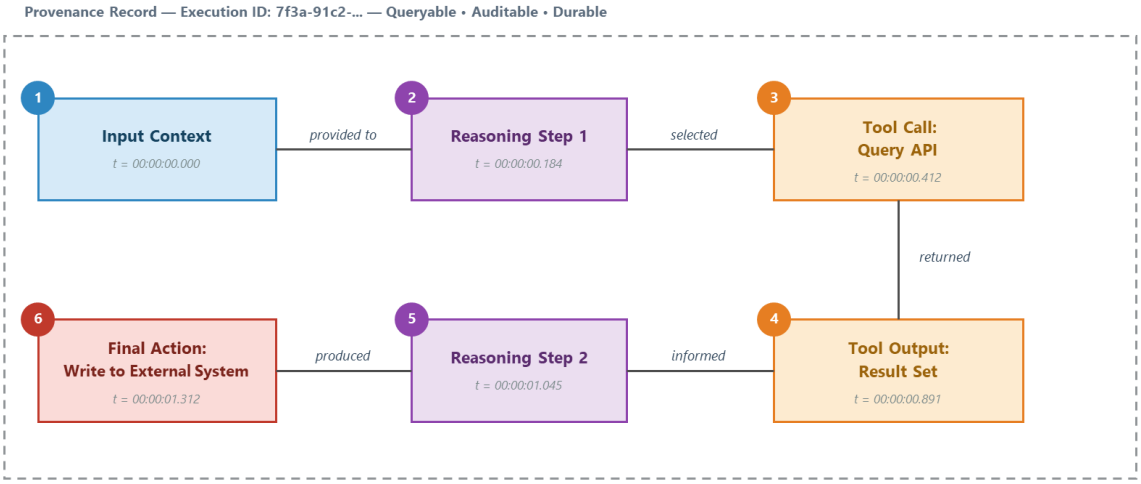

Figure 3. A structured provenance graph for a single agent decision cycle links input context, reasoning steps, tool calls, tool outputs, and final actions into a queryable causal record. This structure enables post-hoc audit, root cause analysis, and compliance reporting that sequential application logs cannot support.

Multi-Agent Coordination and Its Engineering Challenges

Multi-agent systems are distributed systems, and are therefore subject to all the potential modes of failure inherent in distributed systems [6], plus the additional possibility of messaging loss and reordering. Individual agents may fail part way through execution, taking with them partial results, leaving other agents to complete based on only a subset of the information they were supposed to consider and shared memory backends subject to race conditions requiring explicit coordination.

A pattern that emerges frequently when looking at code review pipelines implemented as multi-agent systems is the following: the planner agent splits up the code change under review and sends individual parts to dedicated reviewer agents, and an aggregator agent synthesizes the review based on the results of the individual reviewers. If one of the reviewer agents fails part way through execution, having started but not completed writing its planned review notes into shared memory, the aggregator will read an inconsistent view of the state of the system, and produce a review summary that does not account for the issues that the failed agent was planning to report. The code change will then proceed to be approved even though there were serious issues with it, with no indication that anything had gone wrong. This is the equivalent of a dirty read in the context of databases, and is prevented in databases by having transactions write to a separate set of memory and only commit once they have finished, at which point other transactions can see the updates [7]. The same principle applies to agents: writes should be treated as transactions for the purposes of concurrency control, and the state an agent has written should not be visible to other agents until it has committed.

Redis, for instance, guarantees consistency for transactions on individual keys, but not across multiple keys unless they're enclosed in MULTI/EXEC blocks — it's up to the application to reason about whether that's sufficient for its access pattern, and the same holds for agents.

- **Practical Engineering Takeaways:** Before implementing any multi-agent system with shared state, spend at least thirty minutes documenting what consistency guarantees the agents expect versus what the storage layer actually provides. Skipping this step produces agents that behave correctly in sequence but reveal subtle distributed-system bugs under concurrency — being stateless in each individual operation doesn't help if the system state as a whole lives in shared memory, queues, or a storage backend.

What Production Reliability Actually Requires

A mature agent execution environment needs six reliability properties. Stated plainly, they show just how far current infrastructure is from production-grade:

- **Action safety —** the runtime checks proposed actions against defined safety and correctness constraints before execution, and blocks or flags anything that violates them.
- **Failure-aware execution with semantic classification —** the runtime distinguishes transient failures, persistent failures, and semantic reasoning errors, and applies the right recovery policy to each. Uniform retry isn't an acceptable default.
- **Durable, recoverable execution state —** state is persisted at tool-call-boundary granularity with transactional consistency, so execution can resume from the last checkpoint after any failure.
- **Structured provenance capture —** a queryable causal record for every run, linking input context, reasoning, tool selection, tool output, and final action.
- **Resource governance with enforcement —** tracked, enforced limits on tokens, API calls, memory, and external quota, with defined behavior when those limits are approached.
- **Structured observability —** telemetry compatible with the observability tooling operations teams already use.

None of these require a better foundation model. All of them require software engineering infrastructure that today gets rebuilt from scratch by individual teams, with no shared standards and no established operational patterns.

Practical Steps Engineering Teams Can Take Today

There is no production-grade agent runtime yet, so all of this has to be done inside whatever you are using today. None of it requires a platform rewrite, but it does take real engineering discipline to get right.

- **Validate every tool response against a contract:** before your agent uses any value returned from a tool, make sure it conforms to an expected schema. If it doesn't, fail the tool call. Malformed data that "almost works" will be much harder to debug later.

A minimal version of that wrapper looks like this:

```
from pydantic import BaseModel, ValidationError
class PackageQueryResult(BaseModel):
package_name: str
version: str
known_vulnerabilities: list[str] = []
def call_tool_safely(tool_fn, *args, **kwargs):
raw = tool_fn(*args, **kwargs)
try:
return PackageQueryResult.model_validate(raw)
except ValidationError as e:
# Fail closed: surface a structured, retryable error
# instead of letting the agent reason over malformed data.
raise ToolContractError(tool_fn.__name__, errors=e.errors())
```

- **Tag everything with a correlation ID:** give every agent run a unique correlation ID and record it in every log line. It doesn't take much to implement, and it dramatically simplifies narrowing down what went wrong across services when something breaks.
- **Checkpoint after every tool call:** after each successful call, write a checkpoint to durable storage — Redis, a database, or similar. This lets you resume from the middle of a run instead of the beginning, which speeds up development and is necessary for any long-running production job.
- **Classify failures before retrying:** different failures require different recovery. A network timeout may warrant a retry; a permissions error may require human intervention. Always classify a failure before attempting to recover from it.
- **Cap the step count and explain why execution stopped:** agents should stop when they near their step budget, describe why, and request human assistance where possible — it's much harder to spot the moment an agent has stopped being useful once it's already past it.
- **Make irreversible operations explicitly require human approval:** Classify all operations as read, reversible write, or irreversible write operations. Any irreversible operation must be explicitly approved by a human before it executes — it is one of the cheapest and most effective safety controls available in production.
- **Measure consumption of every run:** track tokens, API calls, and execution time for every run, with alerts when a run is consuming significantly more than usual — while there's still time to stop it.

These measures taken together make agents reliable enough to be used in production without a dedicated runtime, and they can all be implemented in whatever system you already have. It is much easier to make these changes now than to try to retrofit them later when you have much more infrastructure to update.

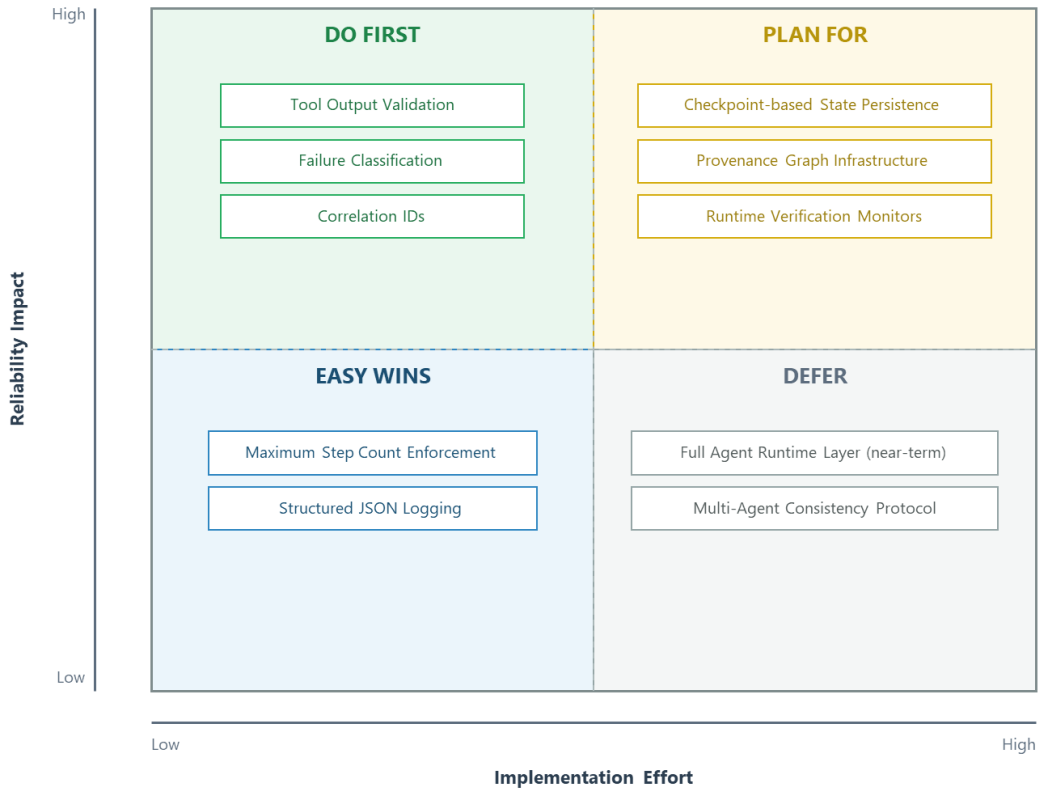

Figure 4. Reliability practices for agent deployments, organized by implementation effort and the value they provide to your agent infrastructure.

The Path Forward Is Systems Engineering

Anyone who has seen a few paradigm shifts in the space has seen this pattern before — cool ideas first, then questions of reliability, then a long process of bringing reliability to the field. Web services and microservices followed this pattern, and so will agents, and so, in turn, will their wrappers.

No invention of new mechanisms is needed — decades of work have demonstrated how to make infrastructural software reliable. TCP/IP protocols made networks reliable [8], and transaction theory made databases reliable [7]; in both cases, systems thinking was applied to novel domains, and that is what is needed for agents. Specifically, whoever makes it their priority to make agent-based infrastructure reliable will reap rich rewards for the entire industry downstream.

Closing: Recommendations, Open Questions, and a Challenge

Three recommendations for teams deploying agents today:

- **Treat tool wrappers as critical infrastructure, not glue code:** every call across agent boundaries stands to fail, waste resources, or exfiltrate data. Wrappers need to be developed with a heavy focus on schema validation, error classification, retries, and telemetry — these are essential ingredients, not nice-to-have features.

- **Set your reliability standard before you need it:** draw up and agree on a reliability design specification for any agent serious about production-critical decisions — observability requirements, resource limits, and failure semantics for the agent and the system as a whole. It takes real effort to draft, but far less than the time and heartache of firefighting after a wrapper has been deployed in anger.
- **Scale provenance investment to consequence:** set provenance requirements in proportion to the potential impact of a mistake. Correlation IDs may be enough for some agents; others, especially those touching safety-critical systems, need far more rigorous auditing and tracing to satisfy trust and safety requirements.

Three open questions that need answers:

What design patterns around failure handling belong at the level of individual agents, and what responsibilities should fall to the wrapper functions instead? Right now, every team is answering that question for itself.

What consistency model should agents that share memory space follow — and won't the answer almost certainly depend on the interaction pattern? There is little established guidance yet for making that call.

How might provenance be standardized to make comparison between agents possible at scale? Without a common schema, provenance records from different frameworks can't be compared, which makes fleet-level audits impractical.

One challenge to pose to future conversations about agents in production: if a software agent executing the seventh of twelve sequential steps discovers an unrecoverable error, what adjustments does that force on the larger process? How is the error made visible to a human operator, how is the process returned to a coherent state, and how long does that take? Until those questions can be answered with specifics, the agent under review is not safe for production, regardless of any other metrics. Correctness and reliability are properties of systems, not of individual components — and for agents, that systems-level work is only just beginning.

References

[1] M. Leucker and C. Schallhart, A brief account of runtime verification (2009), Journal of Logic and Algebraic Programming, 78(5), 293–303

[2] H. Garcia-Molina and K. Salem, Sagas (1987), ACM SIGMOD Record, 16(3), 249–259

[3] Temporal Technologies, Temporal workflow documentation (2024), https://docs.temporal.io

[4] L. Moreau et al., PROV-DM: The PROV Data Model (2013), W3C Recommendation, https://www.w3.org/TR/prov-dm/

[5] W3C PROV Working Group, PROV Overview (2013), https://www.w3.org/TR/prov-overview/

[6] A. S. Tanenbaum and M. Van Steen, Distributed Systems: Principles and Paradigms, 2nd ed. (2007), Prentice Hall

[7] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques (1992), Morgan Kaufmann

[8] V. G. Cerf and R. E. Kahn, A protocol for packet network intercommunication (1974), IEEE Transactions on Communications, 22(5), 637–648

About the Author

Nishil Pathak is an independent software engineering researcher focused on the reliability of production AI agent systems. His work applies principles from operating systems design, distributed systems, and runtime verification to the problem of running autonomous agents safely at scale. The scenarios in this article are composites drawn from patterns observed across multiple production agent deployments, reconstructed to illustrate representative failure modes rather than to report on a single named system.

Figure Attribution: Every figure in this article was made by the author in Inkscape, built from scratch specifically for this piece. Nothing was borrowed, screenshotted, or lifted from anywhere else — no permissions needed, no rights conversations had. They were made alongside the writing and will hopefully find their way into future work too.